Physics 411: Homework 8

1. **FTCS solution of the wave equation:** Consider a piano string of length L, initially at rest. At time $t = 0$ the string is struck by the piano hammer at a point d from the end of the string:

The string vibrates as a result of being struck, except at the ends, $x = 0$ and $x = L$, where its displacement is zero because it is held fixed.

(a) Write a program that uses the FTCS method to solve the complete set of simultaneous first-order equations, Eq. (9.27), for the case $v = 100\,\mathrm{ms}^{-1}$, with the initial condition that $\phi(x) = 0$ everywhere but the velocity $\psi(x)$ is nonzero, with profile

$$\psi(x) = C\frac{x(L-x)}{L^2}\exp\left[-\frac{(x-d)^2}{2\sigma^2}\right],$$

where $L = 1\,\mathrm{m}$, $d = 10\,\mathrm{cm}$, $C = 1\,\mathrm{ms}^{-1}$, and $\sigma = 0.3\,\mathrm{m}$. You will also need to choose a value for the time-step h. A reasonable choice is $h = 10^{-6}\,\mathrm{s}$.

(b) Make an animation of the motion of the piano string using the facilities provided by the `visual` package. There are various ways you could do this. A simple one would be to just place a small sphere at the location of each grid point on the string. A more sophisticated approach would be to use the `curve` object in the `visual` package—see the on-line documentation at `www.vpython.org` for details. A convenient feature of the curve object is that you can specify its set of x positions and y positions separately as arrays. In this exercise the x positions only need to specified once, since they never change, while the y positions will need to be specified anew each time you take a time-step. Also, since the vertical displacement of the string is much less than its horizontal length, you will probably need to multiply the vertical displacement by a fairly large factor to make it visible on the screen.

Allow your animation to run for some time, until numerical instabilities start to appear.

☑ **For full credit** turn in a printout of your program and two snapshots of the animation it produces, taken before and after the instabilities appear.

2. **The Schrödinger equation and the Crank–Nicolson method:**

Perhaps the most important partial differential equation, at least for physicists, is the Schrödinger equation. In this problem you will use the Crank–Nicolson method to solve the full time-dependent Schrödinger equation and hence develop a picture of how a wavefunction evolves over time.

We will look at the Schrödinger equation in one dimension. The techniques for calculating solutions in two or three dimensions are basically the same as for one dimension, but the calculations take much longer on the computer, so in the interests of speed we'll stick with one dimension. In one dimension the Schrödinger equation for a particle of mass M with no potential energy reads

$$-\frac{\hbar^2}{2M}\frac{\partial^2\psi}{\partial x^2} = \mathrm{i}\hbar\frac{\partial\psi}{\partial t}.$$

For simplicity, let's put our particle in a box with impenetrable walls, so that we only have to solve the equation in a finite-sized space. The box forces the wavefunction ψ to be zero at the walls, which we'll put at $x = 0$ and $x = L$, so that the box has size L.

Replacing the second derivative in the Schrödinger equation with a finite difference and applying Euler's method, we get the FTCS equation

$$\psi(x,t+h) = \psi(x,t) + h\frac{\mathrm{i}\hbar}{2ma^2}\bigl[\psi(x+a,t) + \psi(x-a,t) - 2\psi(x,t)\bigr],$$

where a is the spacing of the spatial grid points and h is the size of the time-step. (Be careful not to confuse the time-step h with Planck's constant $\hbar$.) Performing a similar step in reverse, we get the implicit equation

$$\psi(x,t+h) - h\frac{\mathrm{i}\hbar}{2ma^2}\bigl[\psi(x+a,t+h) + \psi(x-a,t+h) - 2\psi(x,t+h)\bigr] = \psi(x,t).$$

And taking the average of these two, we get the Crank–Nicolson form for the Schrödinger equation:

$$\begin{aligned}\psi(x,t+h) - h\frac{\mathrm{i}\hbar}{4ma^2}&\bigl[\psi(x+a,t+h) + \psi(x-a,t+h) - 2\psi(x,t+h)\bigr]\\ &= \psi(x,t) + h\frac{\mathrm{i}\hbar}{4ma^2}\bigl[\psi(x+a,t) + \psi(x-a,t) - 2\psi(x,t)\bigr].\end{aligned}$$

This gives us a set of simultaneous equations, one for each grid point.

The boundary conditions on our problem tell us that $\psi = 0$ at $x = 0$ and $x = L$ for all t and in between these points we have grid points at a, $2a$, $3a$, and so forth. Let us arrange the values of ψ at these points into a vector

$$\boldsymbol{\psi}(t) = \begin{pmatrix}\psi(a,t)\\ \psi(2a,t)\\ \psi(3a,t)\\ \vdots\end{pmatrix}.$$

Then the Crank–Nicolson equations can be written in the form

$$\mathbf{A}\boldsymbol{\psi}(t+h) = \mathbf{B}\boldsymbol{\psi}(t),$$

where the matrices $\mathbf{A}$ and $\mathbf{B}$ are both tridiagonal:

$$\mathbf{A} = \begin{pmatrix} a_1 & a_2 & & & \\ a_2 & a_1 & a_2 & & \\ & a_2 & a_1 & a_2 & \\ & & a_2 & a_1 & \\ & & & & \ddots \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} b_1 & b_2 & & & \\ b_2 & b_1 & b_2 & & \\ & b_2 & b_1 & b_2 & \\ & & b_2 & b_1 & \\ & & & & \ddots \end{pmatrix},$$

with

$$a_1 = 1 + h\frac{\mathrm{i}\hbar}{2ma^2}, \qquad a_2 = -h\frac{\mathrm{i}\hbar}{4ma^2}, \qquad b_1 = 1 - h\frac{\mathrm{i}\hbar}{2ma^2}, \qquad b_2 = h\frac{\mathrm{i}\hbar}{4ma^2}.$$

(Note the different signs and the factors of 2 and 4 in the denominators.)

The equation $\mathbf{A}\boldsymbol{\psi}(t+h) = \mathbf{B}\boldsymbol{\psi}(t)$ has precisely the form $\mathbf{A}\mathbf{x} = \mathbf{v}$ of the simultaneous equation problems we studied earlier in the semester and can be solved using the same methods. Specifically, since the matrix $\mathbf{A}$ is tridiagonal in this case, we can use the fast tridiagonal version of Gaussian elimination that we looked at in Section 6.1.6.

Consider an electron (mass $M = 9.109 \times 10^{-31}$ kg) in a box of length $L = 10^{-8}$ m. At time $t = 0$ the wavefunction of the electron takes the form of a Gaussian wave packet thus:

$$\psi(x,0) = \exp\left[-\frac{(x-x_0)^2}{2\sigma^2}\right]\mathrm{e}^{\mathrm{i}\kappa x},$$

where

$$x_0 = \frac{L}{2}, \qquad \sigma = 1 \times 10^{-10}\,\mathrm{m}, \qquad \kappa = 5 \times 10^{10}\,\mathrm{m}^{-1},$$

and $\psi = 0$ on the walls at $x = 0$ and $x = L$.

(a) Write a program to apply the Crank–Nicolson method to solve the Schrödinger equation for this electron and calculate the vector $\boldsymbol{\psi}(h)$ of values of the wavefunction, given the initial wavefunction above and using $N = 1000$ spatial slices with $a = L/N$. Your program will have to perform the following steps. First, given the vector $\boldsymbol{\psi}(0)$ at $t = 0$, you will have to multiply by the matrix $\mathbf{B}$ to get a vector $\mathbf{v} = \mathbf{B}\boldsymbol{\psi}$. Because of the tridiagonal form of $\mathbf{B}$, this is fairly simple. The ith component of $\mathbf{v}$ is given by

$$v_i = b_1\psi_i + b_2(\psi_{i+1} + \psi_{i-1}). \tag{1}$$

You will also have to choose a value for the time-step h. A reasonable choice is $h = 10^{-18}$ s.

Second you will have to solve the linear system $\mathbf{A}\mathbf{x} = \mathbf{v}$ for $\mathbf{x}$, which gives you the new value of $\boldsymbol{\psi}$. You could do this using a standard linear equation solver like the function `solve` in `numpy.linalg`, but since the matrix $\mathbf{A}$ is tridiagonal a better approach would be to use the fast solver for banded matrices given in Appendix E of the course-pack, which can be imported from the file `banded.py` (which you can find on the web site).

Third, once you have the code in place to solve the equations, extend your program to perform the same operations repeatedly, and hence solve for ψ at a sequence of time-steps with separation h. Note that the matrix $\mathbf{A}$ is independent of time, so it doesn't change from one step to another. You can set up the matrix just once and then keep on reusing it for every step.

(b) Extend your program to make an animation of the wavefunction by displaying the real part of the wavefunction at each time-step. You can use the function `rate` from the package `visual` to ensure a smooth frame-rate for your animation.

As in problem 1 there are various ways you could do the animation. You could just place a small sphere at each grid point with vertical position representing the value of the real part of the wavefunction, or you could use the `curve` object from the package `visual`. Depending on what coordinates you use for measuring x, you may need to scale the values of the wavefunction by an additional constant to make them a reasonable size on the screen. (If you measure your x position in meters then a scale factor of about 10^{-9} works well for the wavefunction.)

(c) Run your animation for a while and describe what you see. Write a few sentences explaining in physics terms what is going on in the system.

☑ **For full credit** turn in a printout of your final program and a snapshot of the animation it produces, along with your explanation of what you discovered upon running the program.

3. **Brownian motion:** Brownian motion is the motion of a particle, such as a smoke or dust particle, in a gas, as it is buffeted by random collisions with gas molecules. Make a simple computer simulation of such a particle (in two dimensions) as follows. The particle is confined to a square grid or lattice $L \times L$ squares on a side, so that its position can be represented by two integers $i, j = 0 \ldots L - 1$. It starts in the middle of the grid. On each step of the simulation, choose a random direction—up, down, left, or right—and move the particle one step in that direction. The particle is doing a "random walk." The particle is not allowed to move outside the square of the lattice—if it tries to do so, choose a new random direction to move in.

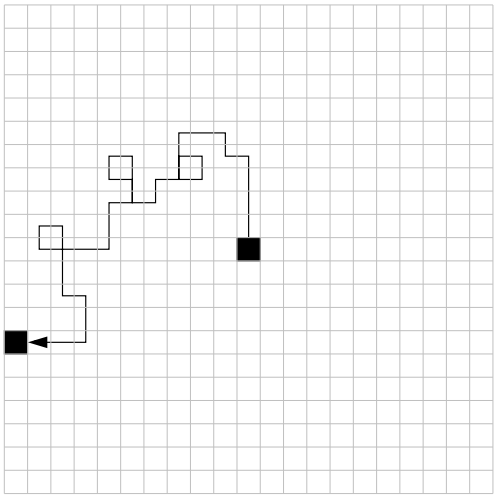
   Write a program to do this calculation for a million steps of the random walk with $L = 1001$ and make an animation on the screen of the position of the particle. (We choose an odd length for the side of the square so that there is one lattice site exactly in the center.)

   Hint: The `visual` package doesn't always work well with the `random` package, but if you import functions from `visual` first, then from `random`, you should avoid problems.

   ☑ **For full credit** turn in a printout of your program.

4. **Diffusion-limited aggregation:** This exercise builds on the previous one on Brownian motion. In this exercise you will develop a computer program to reproduce one of the most famous models in computational physics, *diffusion-limited aggregation* (or DLA for short), invented by UM Professor of Physics Leonard Sander in 1981. There are various

versions of the DLA process, but the one we'll study is as follows. You take a square grid with a single particle in the middle. The particle performs a random walk until it reaches a point on the edge of the square, at which point it "sticks" to the edge, becoming anchored there and immovable:



Then a second particle starts at the center and does a random walk until it sticks either to an edge or to the other particle. Then a third particle starts, and so on. Each particle starts at the center and walks until it sticks either to an edge or to any anchored particle.

(a) Make a copy of the Brownian motion program that you wrote for problem 3. This will serve as a starting point for your DLA program. Modify your program to perform the DLA process on a 101×101 lattice. Now repeatedly introduce a new particle at the center and have it walk randomly until it sticks to an edge or an anchored particle.

You will need to decide some things. How are you going to store the positions of the anchored particles? On each step of the random walk you will have to check the neighboring squares to see if they border on the edge of the square or are occupied by an anchored particle. How are you going to do this? You should also modify your visualization code from the Brownian motion exercise to visualize the positions of both the randomly walking particles and the anchored particles. Run your program for a while and observe what it does.

(b) In the interests of speed, change your program so that it shows only the anchored particles on the screen and not the randomly walking particles. That way you need to update the pictures on the screen only when a new particle becomes anchored. Also remove any `rate` statements that you added to make the animation smooth.

Set up the program so that it stops running once there is an anchored particle in the center of the grid, at the point where each particle starts its random walk. Once there is a particle at this point, there's no point running any longer because any further particles added will be anchored before they can even move anywhere.

Run your program and see what it produces. If you are feeling patient, try modifying it to use a 201×201 lattice and run it again—the pictures will be more impressive but you'll have to wait longer to generate them.

A nice further twist is to modify the program so that the anchored particles are shown in different shades or colors depending on their age, with the shades or colors changing gradually from the first particle added to the last.

(c) **Extra credit:** If you are feeling particularly ambitious, try the following. The original version of DLA was a bit different from the version above—and more difficult to do. In the original version you start off with a single *anchored* particle at the center of the grid and a new particle starts from a random point on the perimeter and walks until it sticks to the particle in the middle. Then the next particle starts from the perimeter and walks until it sticks to one of the other two, and so on. Particles no longer stick to the walls, but they are not allowed to walk off the edge of the grid.

Unfortunately, simulating this version of DLA directly takes a long time—the single anchored particle in the middle of the grid is difficult for a random walker to find, so you have to wait a long time even for just one particle to finish its random walk. But you can speed it up using a clever trick: when the randomly walking particle does finally find its way to the center, it will cross any circle around the center at a random point—no point on the circle is special so the particle will just cross anywhere. But in that case we need not wait the long time required for the particle to make its way to the center and cross that circle. We can just cut to the chase and start the particle on the circle at a random point, rather than at the boundary of the grid. Thus the procedure for simulating this version of DLA is as follows:

i. Start with a single anchored particle in the middle of the grid. Define a variable r to record the furthest distance of any anchored particle from the center of the grid. Initially $r = 0$.

ii. For each additional particle, start the particle at a random point around a circle centered on the center of the grid and having radius $r + 1$. You may not be able to start exactly on the circle, if the circle doesn't pass exactly through a grid point, in which case start on the nearest grid point outside the circle.

iii. Perform a random walk until the particle sticks to another, except that if the particle ever gets more than $2r$ away from the center, throw it away and start a new particle at a random point on the circle again.

iv. Every time a particle sticks, calculate its distance from the center and if that distance is greater than the current value of r, update r to the new value.

v. The program stops running once r surpasses a half of the distance from the center of the grid to the boundary, to prevent particles from ever walking outside the grid.

Try running your program with a 101×101 grid initially and see what you get.

☑ **For full credit** turn in a printout of your program and a snapshot showing the final state of the animation it produces. If you do the extra credit part, turn in your program and a snapshot for that part too.